Undo and Redo Support for Replicated Registers

Leo Stewen
lstwn@mailbox.org
Technical University of Munich
Germany

Martin Kleppmann
martin@kleppmann.com
University of Cambridge
Great Britian

Abstract

Undo and redo functionality is ubiquitous in collaboration software. In single user settings, undo and redo are well understood. However, when multiple users edit a document, concurrency may arise, and the operation history can become non-linear. This renders undo and redo more complex both in terms of their semantics and implementation.

We survey the undo and redo semantics of current mainstream collaboration software and derive principles for undo and redo behavior in a collaborative setting. CRDTs are a tool to allow for concurrent editing of a shared document without requiring central coordination. A simple CRDT is the Multi-Valued Replicated Register, for which we present a novel undo and redo algorithm that implements our identified undo and redo semantics.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords: undo, redo, CRDT, eventual consistency

1 Introduction

Collaborative editing is an essential feature in modern software. Current mainstream collaboration software puts most of the collaborative editing logic in the cloud, at the cost of privacy, control over data and latency [5]. Despite these drawbacks, the power of collaboration is so compelling that users came to accept this trade-off. In recent times, however, there has been a push towards local-first software [5] that aims to bring the benefits of collaboration to software that works with, but does not require the cloud to function. Conflict-Free Replicated Data Types (CRDTs) [8] are the major technological enabler of this movement. They allow for concurrent editing of a shared document without requiring central coordination while still guaranteeing strong eventual consistency. Hence, they are a good fit for local-first software where clients are offline for extended periods of time and need to synchronize their changes when they come back online.

As much as collaboration is a common feature today, so is undo and redo support. Integrating it with CRDTs is not trivial. Although there are some published algorithms for undo and redo in CRDTs (see Section 8), their semantics do not match the behavior of current mainstream applications as detailed in Section 2. However, the algorithms of mainstream applications have not been published or analyzed in research literature and we believe that their semantics are more in line with user expectations.

We aim to fill this gap by providing two contributions. First, we survey the semantics of undo and redo in current mainstream collaboration software and derive principles of undo and redo in a collaborative setting based on this survey. Second, we propose a novel undo and redo algorithm that adheres to the identified semantics. It works on top of Multi-Valued Replicated Registers (MVRs) [10], a core CRDT that is often used as a basic building block to build more complex CRDTs via composition. The intended environment for this algorithm is the Automerge [12] library, which is a JSON CRDT that is based on MVRs.

2 Semantics of Undo/Redo with Multiple Users

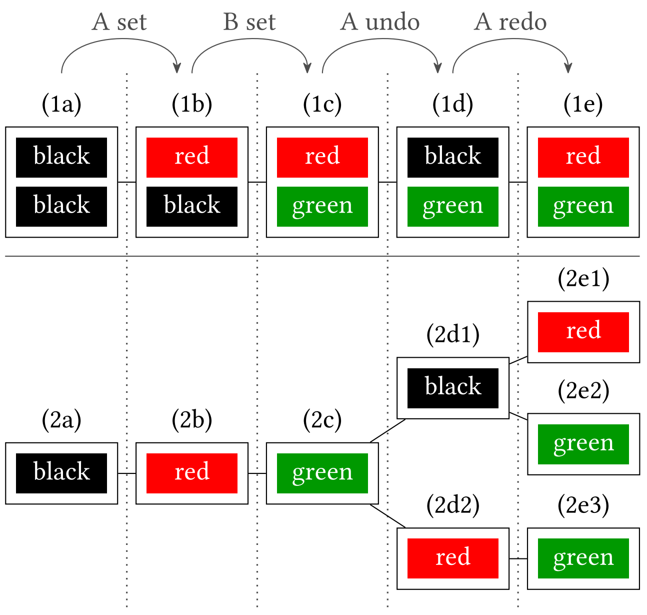

Figure 1. Different semantics of undo and redo with two users A and B. The upper part (1a-1e) depicts behavior in case of two replicated registers and the lower part (2a-2e) shows possible outcomes for one replicated register.

To illustrate the possible semantics of undo and redo with multiple users, we introduce two scenarios. In both scenarios, we consider replicated registers that are used to store the fill color of a rectangle, which exists, for instance, on a slide of a presentation deck. The rectangle's color is stored in a Last-Write-Wins Register (LWWR) [10]. If multiple users concurrently set its color, the register picks one of the concurrently assigned colors arbitrarily as the merged final state. Figure 1 illustrates the two scenarios showing different outcomes of undo and redo over the same sequence of operations: First, *A* sets the color of a rectangle to red and then *B* sets the color to green. Afterwards, *A* performs an undo followed by a redo. The difference between the upper scenario (1a-1e) and the lower scenario (2a-2e) is that in the former, *A* and *B* do not work on the same register: *A* works exclusively on the upper rectangle and *B* on the lower one. In the lower scenario, *A* and *B* collaborate on the same rectangle. All operations are immediately synced to the other user before another change is introduced, avoiding concurrency for now. Hence, both users have seen the same progression of colors.

In the upper scenario (1a-1e), the undo operation by *A* restores the color of the upper rectangle to black and the redo operation restores the color to red. This appears to be the most intuitive behavior of undo and redo as an undo is usually intended to reverse the user's last own change because of a mistake. Moreover, this behavior does not interfere with *B*'s green coloring of the lower rectangle which *A* did not touch at all. Another approach would be that *A*'s undo operation restores the color of the lower rectangle to black and the redo operation restores the color back to green. This would undo the last change of any user at the time of issuing the undo. However, this behavior may be confusing to users especially if the two rectangles are not on the same slide but on different ones. *A* may be surprised to see no immediate effect of her undo because it reverted *B*'s green coloring on a slide *A* is currently not looking at, while actually expecting her last change to be undone. On the other hand, *B* may surprised to learn that her last change was undone for no obvious reason. Therefore, we favor the first approach.

The lower scenario (2a-2e) gives rise to different possible outcomes which may not look intuitive at first sight. *A*'s undo operation can either:

1. Restore the color of the rectangle to black ((2d1) in Figure 1; **local undo**)
2. Restore the color of the rectangle to red ((2d2) in Figure 1; **global undo**)
3. Be not allowed at all (not shown in Figure 1)

We call the first approach *local undo* because it undoes the last operation that was performed by the user who issues the undo. In case operations by other users fall temporally in between the last local operation and the undo operation, their effects are skipped (*B*'s green coloring in (2c)). The second approach is called *global undo* because it undoes the last operation that was performed by *any* user.

The effect of *A*'s subsequent redo operation depends on the used undo behavior. If undo was not allowed, the redo operation is not possible. With global undo behavior, *A*'s redo operation only has one choice: to restore the color of the rectangle back to green (2e3). In case local undo behavior is used, *A*'s redo operation can either:

1. Restore the color of the rectangle to red ((2e1) in Figure 1; **local redo**)
2. Restore the color of the rectangle to green ((2e2) in Figure 1; **global redo**)

Similarly, we call the first approach *local redo* because it redoes the effect of the undone operation from the user who issues both the undo and the redo. The second approach is called *global redo* because it restores to the state prior to the undo operation, regardless of the origin of the operations that constitute this state.

To determine the desired behavior of undo and redo with multiple users, we took inspiration from existing systems which have presumably conducted studies to learn what behavior users expect. Hence, we surveyed the behavior of undo and redo in Google Sheets, Google Slides, Microsoft Excel Online, Microsoft PowerPoint Online, Figma and Miro Boards. For canvas-based applications, namely Google Slides, Microsoft PowerPoint Online, Figma and Miro Boards, the setup is just like the lower scenario in Figure 1 and over the same sequence of operations. For spreadsheet-based applications, namely Google Sheets and Microsoft Excel Online, the behavior of replicated registers is derived from the value of a single cell in the spreadsheet. However, instead of recoloring a rectangle, the value of the cell is incremented in turns by the two users *A* and *B*, starting from zero.

All applications exhibit local undo and global redo behavior except for Miro Boards. It makes undo impossible every time an operation by another user is received, settling on the third undo approach. We suspect that this was a deliberately chosen trade-off between user experience and implementation simplicity in favor of the latter. Hence, we do not consider this behavior any further. We follow the approach of mainstream collaboration software for two reasons: First, the local undo and global redo behavior is consistent with our findings from the two-register scenario in Figure 1. To maintain a consistent user experience, we favor similar semantics regardless of the number of registers involved. Second, global undo implicitly assumes that a user is aware of all remote changes but the strongest assumption we can reasonably make is to assume that she is aware of her own changes. Moreover, global undo poses some additional challenges in an eventually consistent setting that are described in Appendix A. The fact that these challenges do not arise in collaborative software that can afford a central server and online clients, yet none of them chose to implement global undo, makes a strong case to follow their lead. We derive the following principles for undo and redo in a collaborative setting:

- **Local undo**: An undo by a user undoes her own last operation on a register, thereby possibly also undoing operations by other users on that register, if they lie temporally in between her last operation and the undo operation.
- **Global redo**: A redo restores the register to the state prior to its corresponding undo, regardless of the origin of the operations that constitute that state.
- **Undo Redo Neutrality** [13]: Assume a register is in state s. A sequence of n undo operations followed by a sequence of n redo operations should restore state s.

The last principle captures a common usage pattern of undo and redo. Often users want to look at a past version and then restore to the most recent version without changing anything.

3 Background on MVRs

A Multi-Valued Replicated Register (MVR) [10] is a data type from the family of CRDTs [8] that can be assigned a value, overwriting any previous value. When multiple values are concurrently assigned, the MVR retains all values that have not been overwritten. All values currently held by a register are called the register's *siblings*. We assume an operation-based CRDT model [2].

In the previous section we have only discussed LWWRs but they suffer from data loss in case of concurrent updates. To mitigate this, we consider MVRs for the rest of this paper. MVRs are a generalization of LWWRs that expose conflicts and enable the application to pick one of the concurrently written values or to merge them using custom application logic. If the application developer prefers the behavior of LWWRs, it is easy to convert an MVR into a LWWR by picking the sibling with the highest timestamp. For undo and redo behavior, this means that an undo (or a redo) operation may reintroduce conflicts that have been resolved in the past.

An *actor* is a process that can generate updates on an MVR and apply them locally as well as broadcast them to other actors which in turn apply them to their own replica of the MVR. Updates to a register are also referred to as *operations*. We call an operation *local to a replica* if it originated from that same replica. Otherwise, we call it *remote*.

The MVR supports the following functions. Later we will extend this interface with undo and redo operations.

- `get() -> Value[]`: Gets all siblings of the register sorted according to some total order (typically based on a logical timestamp) of the operations.
- `set(value: Option<Value>) -> SetOp<Value>`: Creates a new operation of kind *SetOp* that overwrites the value of the register with the specified value. If no value is provided, the register is cleared, enabling delete functionality. This operation is immediately applied to the local replica and broadcast to all other actors.

Furthermore, each operation has a unique identifier (a logical timestamp) which imposes a total order over all operations. This order is a linear extension of the happens-before relation [6]. We call this identifier the *Operation Id (OpId)*. An example of such an identifier is pair of a local counter and a unique identifier for each actor. For instance, op_3^A denotes an operation by actor A with a local counter value of 3. Whenever a new operation is generated, its counter value is set to one plus the greatest counter value of any existing operation known to the generating replica. This essentially yields a Lamport clock [6] that considers each operation as an event. However, unlike the original Lamport clock, we do not count the receiving or sending of operations as events. This total order is only used for determining the order among siblings in case of concurrent operations but not for determining whether two operations are concurrent or one happened-before the other. This is done by causal dependency tracking via a predecessor relation which we discuss next.

When multiple actors concurrently edit a document, the operation history becomes non-linear. We model operation histories as a directed acyclic graph where each node represents an operation and each edge from node op_2 to node op_1 represents a causal dependency of op_2 on op_1, that is, if op_2 overwrites the register value previously assigned by op_1. We call op_1 a *predecessor* of op_2 and op_2 a *successor* of op_1. Furthermore, we call op_1 an *ancestor* of op_2 and op_2 a *descendant* of op_1 if there exists a directed path from op_2 to op_1. We call op_1 and op_2 *concurrent* if neither is an ancestor of the other. The operations that do not have any successors are called *heads* and the operations that do not have any predecessors are called *roots*. An operation may have multiple predecessors and multiple successors. An operation with multiple successors is the result of concurrency. Whenever an operation has multiple predecessors, we call it a *merge* operation.

Figure 2 shows an example of an operation history which is replicated across two actors A and B. In step (4), $set_1^A(1)$ is the only root and $del_4^A()$ is the only head. $del_4^A()$ is also a merge operation as it has two predecessors, the two heads from step (3). $set_3^A(3)$ and $set_3^B(4)$ are concurrent operations. Our operation history has similarities with git's commit graph which is also a directed acyclic graph, with nodes representing commits rather than operations.

Operations can be delivered in any order and also multiple times because a replica can easily ensure idempotence by keeping track of the applied operations through their OpIds. Each replica buffers operations it receives and delivers them once they are causally ready, i.e., once all ancestors of an operation have been delivered. Unlike a traditional causal broadcast, our algorithm requires storing the operations with a traversable predecessor relationship.

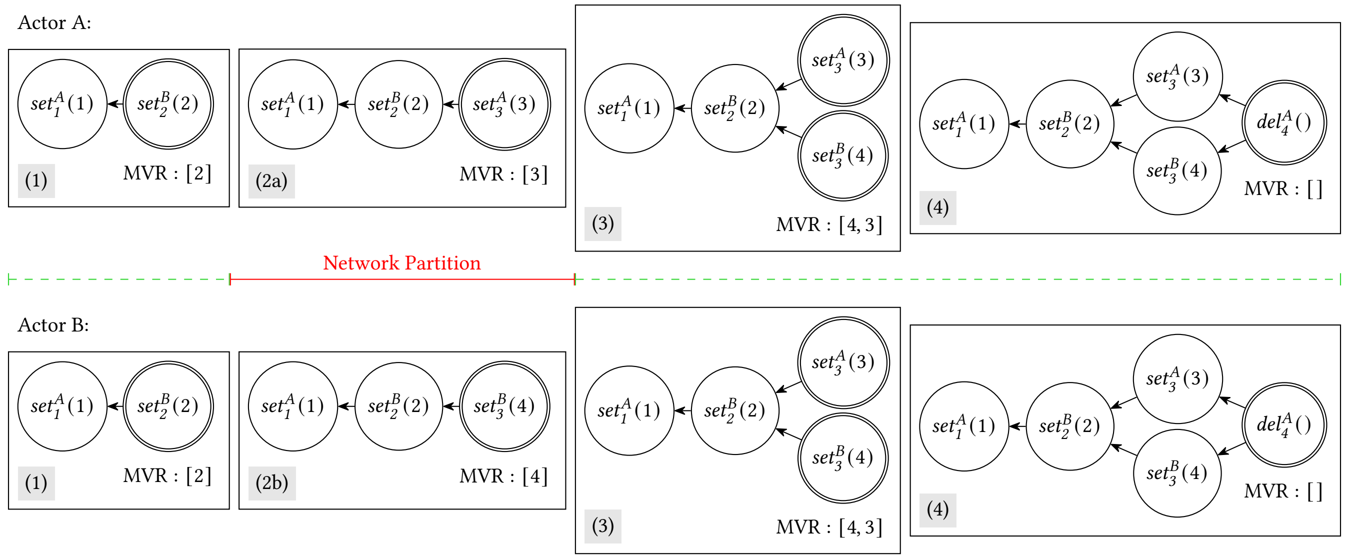

Figure 2. An operation history which is replicated between two actors A and B. Time flows from left to right and is discretized into four different steps depicted by the boxes. At each step the state of the MVR is depicted below the current heads. Heads are denoted as circles with double borders. Due to concurrent operations during a network partition (2a) and (2b), the MVR diverges and causes a conflict (3) which is later reconciled by A's final delete operation (4).

4 Generating Operations for Undo/Redo

So far the MVR supports one kind of operation, the *SetOp*. It is used to enable both setting a value and deleting values from the register. In case of a set operation, the value is supplied in the payload of the *SetOp*. In case of a delete operation, the payload of the *SetOp* is empty. All operations carry an OpId and its set of predecessor OpIds, that represent their causal dependencies. The values produced by these predecessors are the ones that the operation overwrites. The *SetOp* is also called a *terminal operation* because it produces a value (or deletes the register) but does not require a further search through the history to find out about its contribution to the register's values.

We introduce a second operation kind that we call *RestoreOp*. The payload of this operation (in addition to its OpId and predecessors) is the OpId of an ancestor operation which we call *anchor* operation. The effect of this operation is to restore the state of the register to the state immediately before the anchor operation by searching through the operation history. This operation kind is used to implement both undo and redo.

To support undo and redo, we need to introduce some additional state-keeping besides the operation history. We introduce two stacks, one for undo and one for redo, with the usual last-in-first-out semantics. Both stacks exclusively contain operations generated on the local replica and ignore remote operations.

When a local terminal operation is generated, it is pushed onto the undo stack and the redo stack is cleared, which causes the loss of the ability to redo after some sequence of undo operations followed by a terminal operation. This behavior is in line with what most mainstream software does and makes the undo and redo semantics easier to comprehend. Clearing the redo stack ensures that redo has a clear next choice of what to redo. Without clearing the stack, redo (and undo) would become a tree to navigate: after a sequence of undo operations followed by a terminal operation which is undone again, a subsequent redo would have the choice of redoing either the previous undo operation or the later terminal operation. This complexity also exists in a single-actor setting. The vim text editor supports undo trees[1] but it is unusual in this regard. To avoid the user-facing complexity caused by undo trees, we follow the mainstream approach of redo stack clearing.

When the local user performs an undo, we generate a *RestoreOp* that references the terminal operation on top of the undo stack as its anchor and pop it off the undo stack. The generated *RestoreOp* is pushed onto the redo stack to allow it to be redone later. When the local user performs a redo, we generate a *RestoreOp* that references the *RestoreOp* on top of the redo stack as its anchor and pop it off the redo stack. Its anchor is resolved to a terminal operation and pushed onto the undo stack to allow it to be undone later for another time. For redo operations, its *RestoreOp*'s anchor is another *RestoreOp*, hence requiring the algorithm to follow an indirection to resolve to a terminal operation.

This algorithm ensures that the undo stack contains only terminal operations and the redo stack contains only *RestoreOps*. That basically renders a redo as an "undo of an

[1]https://vimhelp.org/undo.txt.html#undo-branches

undo". Finally, we extend the MVR interface with undo and redo functions.

- `undo() -> Option<RestoreOp>`: Provided that the generating replica's undo stack is not empty, this function generates a new operation of kind *RestoreOp* that pops the top operation off the undo stack and references it as its anchor. The new operation is immediately applied to the local replica and broadcast to all other actors.
- `redo() -> Option<RestoreOp>`: Provided that the generating replica's redo stack is not empty, this function creates a new operation of kind *RestoreOp* that pops the top operation off the redo stack and references it as its anchor. Again, the new operation is immediately applied to the local replica and broadcast to all other actors.

5 Applying Operations

Whenever a replica receives a remote operation, it waits for any missing ancestors to arrive first, if necessary, and adds it to its operation history. We introduce the algorithm that determines the current value(s) of the register given an operation history containing *SetOp*s and *RestoreOp*s. We split the discussion into three smaller units: First, we discuss Algorithm 1 that resolves the heads of the operation history to some terminal heads. Second, we present Algorithm 2 that maps the resulting terminal heads to the value(s) of the register. To sort the values in the register appropriately, Algorithm 3 defines a comparison function for the terminal heads that Algorithm 2 utilizes.

Algorithm 1 Resolve Heads to Terminal Heads

```
1:  function (heads)
2:      todo ← ([head, ()] for each head ∈ heads)
3:      termHeads ← ()
4:      while todo is not empty do
5:          [nextOp, opIdTrace] ← todo.shift()
6:          opIdTrace ← (... opIdTrace, nextOp.opId)
7:          if isTerminalOp(nextOp) then
8:              termHeads.push([nextOp, opIdTrace])
9:          else                                  ▷ current is a RestoreOp
10:             anchor ← nextOp.anchor
11:             for each pred ∈ anchor.predecessors do
12:                 todo.push([pred, opIdTrace])
13:     return termHeads
```

Algorithm 1 takes the current set of heads of the operation history and returns a list of (*TerminalOp*, *OpIdTrace*) pairs which we refer to as terminal heads. The *OpIdTrace* is a list of OpIds from the operations that have been visited along the path from a head to a terminal operation. The *shift()* method on a list pops the first element off the list and returns it. The *push()* method on a list appends an element to the list.

The algorithm proceeds as follows: If a head is a *SetOp* (i.e., a terminal operation), it is a terminal head and added to the terminal heads list together with the *OpIdTrace* that contains only the OpId of the *SetOp*. If a head is a *RestoreOp*, the algorithm traverses the operation history by considering its anchor's predecessors iteratively until terminal operations are encountered. While traversing the operation history, every encountered OpId is added to the *OpIdTrace* which is passed along to the next iteration. When the search stops at a terminal operation, the terminal operation and the *OpIdTrace* are added to the terminal heads list as a pair. The *OpIdTrace*'s last element is always an OpId from a terminal operation and any preceding element is an OpId from a *RestoreOp*. The processing order of heads and predecessors in the *todo* list is not relevant for correctness as every terminal head is sorted later in Algorithms 2 and 3.

The reason for the differentiation between terminal operations and *RestoreOp*s is that a *RestoreOp* may contribute multiple values to the register due to concurrency in the operation history. For instance, if at any iteration step a merge *RestoreOp* having k predecessors is processed, all k predecessors are considered by the algorithm and they may produce k or more siblings. In a nutshell, a *RestoreOp* obtains its *values* by iteratively following the predecessor relationship but the *ordering* among the siblings produced by this traversal is determined by the *OpIdTrace* as we will see when discussing Algorithm 3.

Algorithm 2 Resolve Terminal Heads to Value(s)

```
1:  function (termHeads)
2:      sortedHeads ← sort(termHeads) desc using Alg. 3
3:      values ← ()
4:      for each head in sortedHeads do
5:          headOp ← head[0]
6:          if (headOp.value ≠ None) then
7:              values.push(headOp.value)
8:      return values
```

Algorithm 2 sets the value(s) of the register given a list of terminal heads consisting of pairs of terminal operations and *OpIdTrace*s. First, it sorts the terminal heads by their *OpIdTrace* in descending order in line 2 using Algorithm 3. Then, it filter maps the sorted terminal heads to their value(s) in lines 3-7. In case of a deletion (*SetOp* with no value supplied), the corresponding terminal head is skipped and no value is produced.

Algorithm 3 provides the comparison logic of two *OpIdTraces* when sorting the list of terminal heads in line 2 of Algorithm 2. The *zip()* function takes two lists and returns a list of pairs where the first element of the pair is from the first list and the second element is from the second list. If the lists are of different lengths, the longer list is truncated to the length of the shorter list. The iteration in lines 3-7 always

Algorithm 3 Comparison Function

```
1: function ($aOpIdTrace$, $bOpIdTrace$)
2:     $zipped \leftarrow zip(aOpIdTrace, bOpIdTrace)$
3:     for each ($aOpId$, $bOpId$) in $zipped$ do
4:         if $aOpId < bOpId$ then
5:             return $Less$
6:         else if $aOpId > bOpId$ then
7:             return $Greater$
```

terminates before finishing with its last iteration because of the nature of the *OpIdTraces*. If two terminal heads stem from different heads they already differ in their first element (the OpId of the head). If two terminal heads stem from the same head (which then must be a *RestoreOp*), they might share a common path but their paths eventually diverge for some elements. The *zip()* function's truncation does not harm this property as an *OpIdTrace*'s last element is always an OpId from a terminal operation which cannot occur on the same position of another, *longer OpIdTrace*.

Figure 3 shows an operation history that is replicated between two actors A and B that contains undo and redo operations. Table 1 shows the internal node states during the time steps from Figure 3. It demonstrates the stack maintenance introduced in Section 4 and shows the register's values produced by Algorithms 1 to 3 which are applied on top of the respective operation histories at the time steps. Steps (2a) and (2b) point out how two concurrent undo operations are resolved. $undo^{A}_{5}\binom{A}{3}$ produces the value 2 with an *OpIdTrace* of $[{}^{A}_{5},{}^{B}_{2}]$. $undo^{B}_{5}\binom{B}{4}$ produces the values 3 with an *OpIdTrace* of $[{}^{B}_{5},{}^{B}_{3}]$ and 4 with an *OpIdTrace* of $[{}^{B}_{5},{}^{A}_{3}]$. Sorting by these *OpIdTraces* yields the register's values [3,4,2] after syncing in (2b).

The concurrent $undo^{B}_{7}\binom{B}{2}$ and $set^{A}_{7}(6)$ operations at (4) highlight the requirement of the *OpIdTrace*. The $undo^{B}_{7}\binom{B}{2}$ operation produces the value 1 with an *OpIdTrace* of $[{}^{B}_{7},{}^{A}_{1}]$. The $set^{A}_{7}(6)$ operation immediately yields the value 6 with an *OpIdTrace* of $[{}^{A}_{7}]$. If there was no *OpIdTrace* and instead the OpIds of the terminal operations were used, the values produced by the undo operation would always lose against the concurrent set operation because the OpIds of the terminal operations are always smaller than the OpId of the concurrent set operation due the traversal of the predecessor relation. To fix this unfairness, the *OpIdTrace* is employed to sort values produced by *different heads* by their head's OpIds and values produced by the *same head* by their first deviation in the *OpIdTrace*.

Finally, at (4) A is giving up its ability to redo because its terminal operation $set^{A}_{7}(6)$ clears the redo stack, rendering A unable to redo its previously undone $set^{A}_{3}(4)$ operation.

6 Optimization

We can make use of two observations to optimize the algorithm: First, the predecessors of an operation remain constant after their generation. Incoming concurrent operations may have existing operations as their predecessors but they never update the predecessors of existing operations. Second, a *RestoreOp*'s anchor is also fixed at generation time. As explained in Section 7, there are degenerate cases in which after following the anchor and resolving its predecessors, one predecessor is another *RestoreOp* and the same pattern repeats for a few iterations, leading to a non-constant runtime. Yet, due to the two key observations above, we are resolving the exact same values for a *RestoreOp* which we have already resolved at the time it was the head of the operation history. Hence, we can harness caching to improve the runtime at the expense of an increased memory overhead.

For each *RestoreOp* we cache the values produced by it in sorted order. As soon we have stored the sorted order, we can even drop the *OpIdTrace* and rely on a *stable* sort algorithm to include the sorted values of a *RestoreOp* at the right position with just the *OpIdTrace* up to the *RestoreOp*'s OpId. This works because if a *RestoreOp* produces multiple values, the part of the *OpIdTrace* going beyond the *RestoreOp* is responsible for sorting the values it produces. Storing the values in sorted order makes that part of the *OpIdTrace* redundant. Only the first part of the *OpIdTrace* up to the *RestoreOp* is important for figuring out the relation of the cached values with other values contributed by other operations. However, this requires the sorting algorithm to be stable.

7 Evaluation

To evaluate the correctness of our algorithm, we demonstrate that it converges, and we check the behavior of our reference implementation with a suite of unit tests. To evaluate its performance, we analyze its runtime complexity for relevant undo and redo scenarios and provide benchmarking results. Finally, we discuss its integration with Automerge.

Correctness. We demonstrate that any two replicas that have received the same set of operations converge to the same state. The same set of operations produce the same operation history on both replicas over which the algorithms from Section 5 are applied. Since the algorithms are deterministic and do not depend on the order in which operations were received, they produce the same value(s) on both replicas, given that they terminate. Algorithm 2's termination is trivial since it sorts and then maps a finite set of operations. The termination of the comparison function from Algorithm 3 is guaranteed by the fact that *OpIdTraces* are finite (due to the operation history being finite). Algorithm 1's termination is a bit more involved. If all heads are terminal operations, termination is trivial as in the while loop, the length of the *todo* list decreases by one in every iteration and no element is inserted. When dealing with *RestoreOps*,

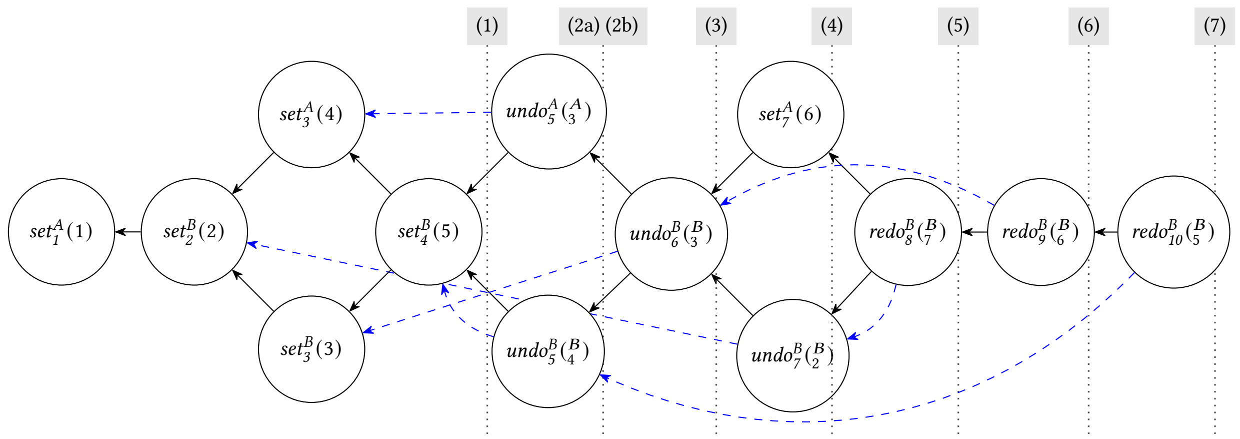

Figure 3. An operation history which is replicated between two actors A and B that includes undo and redo operations. For clarity, undo and redo operations are not labeled as *RestoreOps*. Their respective anchor operations are indicated by the dashed blue arrows and denoted in parenthesis. The dotted lines indicate time steps during which we show the nodes' states in Table 1.

Time Step	(1)	(2a)	(2b)	(3)	(4)	(5)	(6)	(7)
$undoStack^A$	$[s_1^A(1), s_3^A(4)]$	$[s_1^A(1)]$	(2a)	(2a)	$[s_1^A(1), s_7^A(6)]$	(4)	(4)	(4)
$redoStack^A$	[]	$[rs_5^A({}^A_3)]$	(2a)	(2a)	[]	(4)	(4)	(4)
$register^A$	[5]	[2]	[3,4,2]	[2]	[1,6]	[2]	[3,4,2]	[5]
$undoStack^B$	$[s_2^B(2), s_3^B(3), s_4^B(5)]$	$[s_2^B(2), s_3^B(3)]$	(2a)	$[s_2^B(2)]$	[]	(3)	(2a)	(1)
$redoStack^B$	[]	$[rs_5^B({}^B_4)]$	(2a)	$[rs_5^B({}^B_4), rs_6^B({}^B_3)]$	$[rs_5^B({}^B_4), rs_6^B({}^B_3), rs_7^B({}^B_2)]$	(3)	(2a)	(1)
$register^B$	[5]	[3,4]	[3,4,2]	[2]	[1,6]	[2]	[3,4,2]	[5]

Table 1. The internal states of the nodes' undo and redo stacks and the values of the register at the time steps from Figure 3. A set operation is abbreviated by *s* and a restore operation by *rs* with its anchor operation denoted within parenthesis. In case a cell contains a reference to another time step, the value is exactly the same as in that respective step. The difference between (2a) and (2b) is that in (2a) both replicas have not yet synced whereas in (2b) they have exchanged their operations. At all other time steps the operations are assumed to be synced.

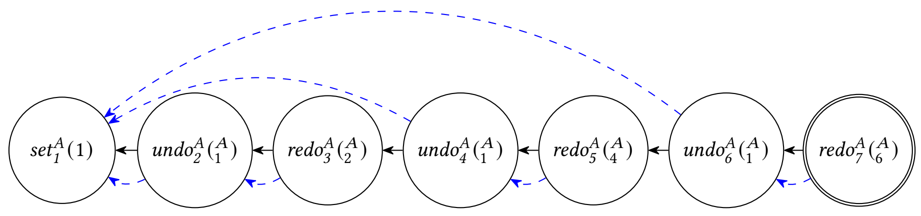

Figure 4. A sequence of alternating undo-redo operations of length 3 (counting one undo-redo-pair as one). In this case, the algorithm's run time is not constant but linear in the length of the sequence.

in every iteration of the while loop the length of the *todo* list also decreases by one but it also increases by the number of predecessors of the anchor operation. Yet, due to the operation history being finite, cycle-free and Algorithm 1 going only backwards in time but never forwards (by following the predecessor relation in case of *RestoreOps*), termination is guaranteed. Therefore, the algorithm converges to the same state on both replicas.

The reference implementation is written in Typescript and consists of around 350 lines of code (excluding comments and blank lines), implementing an MVR with undo and redo functionality as described in Sections 3 to 5. The implementation is tested against a set of unit tests that covers important scenarios, both in single-user and multi-user cases. In total there are 30 unit tests, comprising of around 640 lines of test code (also excluding comments and blank lines).

Performance. Algorithm 2's runtime is entirely driven by the complexity of sorting the terminal operations and Algorithm 3's runtime is linear in the length of the shorter *OpIdTrace*. To assess Algorithm 1's runtime, we look at the runtime of resolving the different operation kinds, *SetOp* and *RestoreOp*. To ease the analysis, we ignore any complexity arising from dealing with data structures like maps and lists but focus on the iterations of the while loop in lines 4-12. In case of a *SetOp*, the runtime is constant as the while loop does not add an element to the *todo* list but just removes the *SetOp* from the list.

For *RestoreOp*s, we push all k predecessors of the referenced anchor operation to the *todo* list and therefore have to process k more operations. Some of these k operations may be *SetOp*s which end the search but some may be *RestoreOp*s themselves. Then, the process repeats until a terminal operation is found or roots are reached that cannot add any predecessors to the *todo* list. The amount of predecessors depends on the degree of concurrency in the system. We restrict the analysis to linear operation histories and therefore assume that there is just one predecessor of a *RestoreOp*. Nevertheless, the following arguments can still be applied for each branch introduced by a predecessor of a *RestoreOp*, albeit with a shorter length.

In most practical editing scenarios, the resolution of a *RestoreOp* head to a value happens in constant time. For example, if a user performs n undo operations (to see prior document states) followed by $m \leq n$ redo operations (to return to one of the document states), the resolution of each undo and redo operation happens in constant time, provided that the anchor operations' predecessor of the n undo operations are all terminal operations. In that case each undo operation resolves to a terminal operation in two iterations (one for its *RestoreOp* and its anchor's predecessor (a terminal operation)) and each redo operation resolves to a terminal operation in three iterations (one for its *RestoreOp*, its anchor's predecessor (an undo operation) and that undo operation's predecessor of its anchor (a terminal operation)).

However, there are scenarios where the resolution of a *RestoreOp* head to a value is not constant but linear in the length of the (per assumption linear) operation history. Figure 4 shows such a scenario where a single actor produces an operation history of alternating undo-redo operations. For resolving the head of a sequence of alternating undo-redo operations of length n (counting one undo-redo-pair as one), the algorithm has to iterate n times to find the terminal operation. Yet, only the resolution of a redo operation suffers from this whereas the resolution of an undo operation

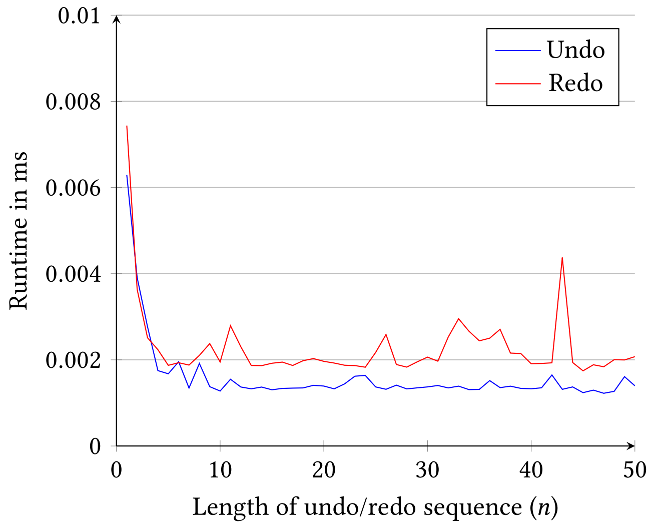

Figure 5. Runtime of the last undo (redo) operation in a sequence of n consecutive set operations followed by n undo operations (n consecutive set operations followed by n undo operations followed by n redo operations).

still happens in constant time. The same effect can be created by alternating set and undo operations. Yet in practice, this is not a real concern as we expect the lengths of such degenerate sequences to be limited as undo and redo is a human-facing feature. More importantly, we think that neither alternating undo and redo nor set and undo operations is a common use case.

Benchmarks of our reference implementation support these findings. Figure 5 and Figure 6 measure the runtime in two different scenarios for undo and redo. Each data point shows the mean runtime over 1024 runs. The benchmarking was conducted on a 2019 MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor and Node version 18.17.1.

Figure 5 covers the undo-redo-neutrality principle from Section 2. For undo operations, the mean runtime of generating and applying an undo operation is shown with $n - 1$ preceding undo operations which are in turn preceded by n set operations, for $n \in \{1, 2, \ldots, 50\}$. For redo operations, the undo sequence is extended by n redo operations. Similarly, it shows the mean runtime of generating and applying the last redo operation. Both undo and redo runtimes are mostly constant and therefore independent of the length of the sequence. We suspect that the initial spike of both undo and redo is due to Node's just-in-time compilation not yet having optimized the code. The outlier for redo operations at $n = 43$ may be explained by a garbage collection pause. Redo operations take slightly longer than undo operations for two reasons. First, a redo operation resolves its anchor operation to a terminal operation and puts it onto the undo stack for possibly undoing it later again, whereas an undo operation just puts its *RestoreOp* onto the redo stack. Second,

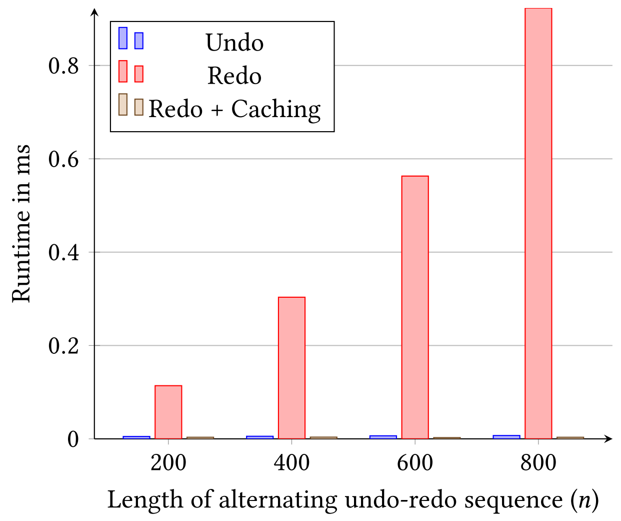

Figure 6. Runtime of the last undo/redo operation in a sequence of alternating undo-redo operations of length n (counting one undo-redo-pair as one; see Figure 4).

resolving to a terminal operation takes three iterations for a redo operation as opposed to just two iterations for an undo operation.

Figure 6 illustrates the mean runtime of the last undo or redo operation of a sequence of alternating undo-redo operations of length n for $n \in \{200, 400, 600, 800\}$. This scenario resembles Figure 4, but with longer sequences, and measures the time to generate and apply the head of the sequence. As previously discussed, the runtime of undo operations is constant in this scenario and not affected by the length of the sequence. The runtime of applying redo operations grows linearly with the length of the sequence. With $n = 800$, applying the redo head takes 0.9 milliseconds. We believe that this does not impact the user experience negatively, since the user will not notice a delay of less than a millisecond and the use of a compiled language like Rust may further reduce the runtime. Finally, we consider an alternating undo-redo sequence of length 800 irrelevant in practice.

Integration with Automerge. Since Automerge stores a monotonically growing set of operations, efficient compression of the serialized operation set is an important property. Our proposed solution models both undo and redo with a single operation kind whose extra payload consists only of a single OpId. Moreover, the *RestoreOp* may also be useful to support global undo behavior in addition to local undo in the future. Since every undo and redo creates a new *RestoreOp* that is added to the set, the undo and redo stacks are implicitly stored within the operation set and can be reconstructed upon loading the document to support undo and redo across sessions without any additional persistency overhead.

Moreover, if an application developer decides not to use undo and redo functionality, the value resolution algorithm will be the same as without undo and redo functionality, thereby not incurring any overhead and making users of the library only pay for what they use. In terms of space complexity, the additional overhead is the introduced state of the undo and redo stacks which can be disabled if not needed and are otherwise just a fraction of the total number of operations of the register and can be bounded if arbitrary undo depth support is not required.

Finally, we remark a possibility to prune the history under a special circumstance. Imagine a user undoing a few times and then redoing the same amount of times just to see the document in the past and then return to the original state. We call such a sequence of undo and redo operations effect-free. Given that all participating actors have seen the same operations and made no changes to the register within the effect-free sequence, the sequence can be pruned (garbage collected) from the history. However, before doing so causal stability [2] must be reached, that is, all actors must have sent an operation whose predecessor set contains the highest OpId of the effect-free sequence.

8 Related Work

The terms local and global undo were introduced by the Operational Transformation literature in the context of collaborative text editing [11, 9, 1], but this distinction is less established in the context of CRDTs.

A common approach to undo and redo in the CRDT literature is to attach a counter (sometimes called *degree* [14] or *undo length* [3, 16]) to each operation. Some approaches [14, 7] initialize the counter to 1 and decrement it on each undo and increment it on each redo. If the counter is positive, the operation is visible, otherwise it is invisible. Other approaches [3, 16] initialize the counter to 0 and increment it on each undo and redo. If it is even, the operation is visible, otherwise it is not. When merging conflicting counters of an operation, the maximum is used. Yet, counter-based approaches handle undo only for a single operation and do not allow skipping over remote operations, which is required for local undo behavior as we pointed out in Section 2. To illustrate this we consider Figure 7. It resembles the single-register scenario in Figure 1 but instead of A redoing in the last step, B issues an undo. Figure 7 shows the correct outcomes according to local undo behavior. Applying counter-based approaches to this scenario poses some challenges. A's undo would render both A's and B's *SetOp* invisible to achieve the correct black coloring of the rectangle (step (4)). B's subsequent undo requires to turn A's *SetOp* visible again to obtain the correct red coloring (step (5)). To achieve this, B's *undo* must cause the effect of a *redo* of A's *SetOp*. Since this simple example is already difficult (and not even considering concurrency), we believe that counter-based approaches are not suitable for implementing local undo.

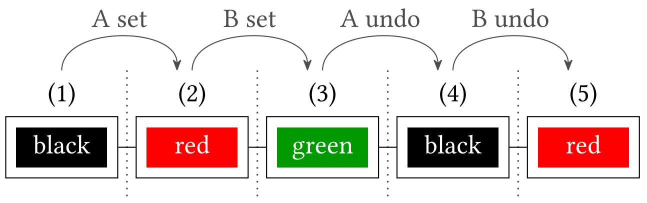

Figure 7. A simple but challenging scenario for counter-based approaches if local undo behavior is desired.

Yu et al. [15] deal with selective undo in the context of strings. Selective undo is a form of undo which allows a user to undo any operation, regardless of *where and when* it was generated. In contrast, global undo allows undoing a remote operation as well, but it only allows undoing in reverse chronological order. A later work of Yu et al. [16] describes a generic undo mechanism, but it only applies to state-based CRDTs, is assuming global undo behavior, and focusses on concurrent undo and redo of a single operation. It does not address the issue of how to order operations on the undo stack with global undo.

Finally, [4] is a rather theoretical work that points out limitations of undo for set and counter CRDTs. Martin et al. [7] discuss undo in the context of XML-like documents. To the best of our knowledge, Brattli et al. [3] is the only work that also discusses undo and redo in the context of replicated registers, but they assume global undo behavior.

9 Conclusion

We tested the semantics of undo and redo of existing collaboration software, and found that almost all use local undo. We then presented a novel algorithm for local undo on an MVR by traversing operation histories represented as directed acyclic graphs. The Automerge CRDT implementation already stores such an operation graph to allow past states of a document to be inspected; therefore our algorithm does not incur any additional overhead in this context. The implementation of our algorithm is currently a standalone prototype written in Typescript, but we plan to integrate it into Automerge in the future. We believe that our algorithm can also be generalized from a single MVR to more complex CRDTs, for example by treating every key in a map and every element in a list as a MVR, and by using (respectively) one shared undo and redo stack for all MVRs in the entire data structure.

References

[1] Gregory D Abowd and Alan J Dix. 1992. Giving undo attention. *Interacting with computers*, 4, 3, 317–342.

[2] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2017. Pure operation-based replicated data types. *arXiv preprint arXiv:1710.04469*.

[3] Eric Brattli and Weihai Yu. 2021. Supporting undo and redo for replicated registers in collaborative applications. In *18th International Conference on Cooperative Design, Visualization, and Engineering* (CDVE 2021). Springer LNCS volume 12983, (Oct. 2021), 195–205. DOI: 10.1007/978-3-030-88207-5_19.

[4] Stephen Dolan. 2020. Brief announcement: the only undoable CRDTs are counters. In *39th Symposium on Principles of Distributed Computing* (PODC 2020). ACM, (Aug. 2020), 57–58. eprint: 2006.10494. DOI: 10.1145/3382734.3405749.

[5] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. 2019. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 154–178.

[6] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21, 7, (July 1978), 558–565. DOI: 10.1145/359545.359563.

[7] Stéphane Martin, Pascal Urso, and Stéphane Weiss. 2010. Scalable XML collaborative editing with undo. In *On the Move to Meaningful Internet Systems (OTM)*. Springer LNCS volume 6426, (Oct. 2010), 507–514. eprint: 1010.3615. DOI: 10.1007/978-3-642-16934-2_37.

[8] Nuno Preguiça. 2018. Conflict-free replicated data types: an overview. *arXiv preprint arXiv:1806.10254*.

[9] Matthias Ressel and Rul Gunzenhäuser. 1999. Reducing the problems of group undo. In *Proceedings of the 1999 ACM International Conference on Supporting Group Work*, 131–139.

[10] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. A comprehensive study of convergent and commutative replicated data types. Tech. rep. Inria–Centre Paris-Rocquencourt; INRIA.

[11] Chengzheng Sun. 2000. Undo any operation at any time in group editors. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, 191–200.

[12] The Automerge Contributors. 2023. *Automerge CRDT Library*. https://github.com/automerge.

[13] Evan Wallace. 2019. How Figma's multiplayer technology works. Retrieved Sept. 4, 2023 from https://web.archive.org/web/20230904085717/https://www.figma.com/blog/how-figmas-multiplayer-technology-works/.

[14] Stephane Weiss, Pascal Urso, and Pascal Molli. 2010. Logoot-Undo: distributed collaborative editing system on P2P networks. *IEEE Transactions on Parallel and Distributed Systems*, 21, 8, (Aug. 2010), 1162–1174. DOI: 10.1109/TPDS.2009.173.

[15] Weihai Yu, Luc André, and Claudia-Lavinia Ignat. 2015. A CRDT supporting selective undo for collaborative text editing. In *15th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems* (DAIS 2015). Springer LNCS volume 9038, (June 2015), 193–206. DOI: 10.1007/978-3-319-19129-4_16.

[16] Weihai Yu, Victorien Elvinger, and Claudia-Lavinia Ignat. 2019. A generic undo support for state-based CRDTs. In *23rd International Conference on Principles of Distributed Systems* (OPODIS 2019) Article 14. Dagstuhl LIPIcs, (Nov. 2019). DOI: 10.4230/LIPIcs.OPODIS.2019.14.

A Challenges with Global Undo

Although we argued against the behavior of global undo, we put forward some considerations if using it in an eventually consistent setting that allows for operations to arrive in causal (or even arbitrary) order.

A.1 Undo Order

In general, an undo mechanism requires some order to determine which operation(s) to undo next. While this does not have to be a total order and one could come up with a scheme how to undo a partial order, the eventual consistent setting poses a technical limit: Suppose there is a sequence of undo operations and the first undone operation has timestamp t_f and the last undone operation t_l. Then, an operation is delivered out-of-order with timestamp t_m such that $t_l < t_m < t_f$ according to the (partial) order. This means that *if* the out-of-order operation had been delivered earlier it had also been undone by the undo sequence. We call such operations *out-of-order delivered and behind the undo cursor*. This leaves the question of how to treat these operations. A potential solution is to turn the undo stack into a priority queue and then undo the operation at the time of arrival at the undoing actor but that may cause hard to understand jumps in time for users. Another option is to ignore these operations but that poses the issue of how to deal with them upon subsequent redos. We are not convinced of any good solution here and think of this as another, rather technical reason to avoid global undo behavior in eventually consistent settings.

Nevertheless, we briefly sketch a scheme how to undo a partial order. The idea is to use multiple cursors that undo in parallel whenever the partial order cannot compare operations (due to concurrency). After a merge operation is undone, assume there are multiple branches that are candidates to undo next. For each branch a cursor pointing to the next operation that is due for undo is created and it initially points to the respective tip of the branch. Upon undo, the MVR is populated with all values contributed by the operations referenced by the cursors, and the cursors are advanced to point to their predecessors. Different strategies may be employed to coordinate the cursors in case any cursor hits a common ancestor or another merge operation.

If the increased complexity from the cursor handling is undesired, the total order imposed by the operation ids could also be used. However, the out of order delivery issue remains for both total and partial orders. In contrast, local undo restricts the undoable operations to the subset of the actor's own operations which are always delivered in order at each actor, and thereby does not suffer from this problem.

A.2 Metadata Overhead

Local undo partitions the set of operations into disjoint sets, one set for each actor. Since an actor can only undo her own operations, it cannot happen that concurrently the same operation is undone multiple times, or that the same operation is both undone and redone by different actors. While it certainly is possible to design some data structure to account for this, it increases the complexity and overhead of the algorithm.